APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: **REAL-TIME STREAMED DATA DOWNLOAD SYSTEM AND METHOD**

APPLICANT: **JUAN C. ALVARADO AND JOHN M. ABNEY, III**

"EXPRESS MAIL" Mailing Label No. **EK857509614US**

Date of Deposit December 18, 2000

I hereby certify under 37 CFR 1.10 that this correspondence is being deposited with the United States Postal Service as **"Express Mail Post Office to Addressee"** with sufficient postage on the date indicated above and is addressed to Box Patent Application, Commissioner for Patents, Washington, D.C. 20231.

Signature Mary L. Thompson

Typed or printed name of person signing this certificate

MARY L. Thompson

REAL-TIME STREAMED DATA DOWNLOAD SYSTEM AND METHOD

Technical Field

This invention relates in general to the field of data transmission, and in particular to the real-time streamed download of data files over a computer network.

Copyright Notice

Background of the Invention

Data transmission is one of the most important tools in many modern businesses. The ready availability of data is often the key to obtaining and maintaining a competitive edge. It can be the kernel of an enterprise.

Many businesses have their very foundation in the collection and dissemination of data. These enterprises range from the medical fields to aviation, from weather forecasting, to petroleum exploration and production.

Efficient and timely transmission of data is critical to the Petroleum Exploration and Production (E & P) industry. E & P activities are extraordinarily expensive undertakings which often take place in locations that are remote and distant from the offices where decisions are made. To maximize the value obtained from such endeavors, data is collected using a variety of surveying methods. These include land and offshore seismic surveys which are vast collections of multi-dimensional

data, wireline well-logging in which data is collected from an electronic instrument lowered into a well, and measurements collected during the drilling operation itself.

Usually, if not always, the data acquired in from an E & P operation, be it seismic surveying, wireline well-logging, or logging while drilling, requires substantial processing before it is useful to make decisions. Such processing may include depicting the data graphically on a graphics workstation or executing one or several data interpretation programs. It is useful for that processing to occur concurrently with the acquisition of the data and transmission of the data from the field location to the location where the data is used, e.g., a data interpretation center or the headquarter of an oil company.

U.S. Pat. No. 5,864,772 describes a system in which petrophysical data collected at a data acquisition site is transmitted in near real time to a remote location. Near real time data transmission refers herein to transmission of data concurrently with data acquisition so that the acquired data is available for viewing or other processing at a remote location nearly at the same time as it is being acquired.

The World Wide Web and the HTTP protocol are designed with the goal of data delivery from a web server to a web browser.

In a standard web server to web browser communication a web page written, for example, in html is transmitted from a server computer using the HTTP protocol over the Internet to a web browser running on a client computer. The web browser interprets the web page and renders it on a screen on the client computer. The standard web environment further allows for file transfer from the server to the client. For example, a web page may have a hyperlink to a document stored on the server. By clicking on the hyperlink a user may cause the transfer of the file from the server to the client and either that the document is opened in a rendering program such as Adobe Acrobat or saved to a disk file.

It would be possible to extend the standard web technology to support real-time data transmission from the server side using CGI scripts, servlets or server

scripts. In such an extension the received data would be stored in real-time by most web browsers. Without client-side custom software, the default behavior of the browser prevents external components from accessing the downloaded data until that data has been completely received. For that reason, real-time applications are not properly launched and data streams cannot be routed to non-file destinations, for example, digital gauges shown within the browser window.

RealNetworks Inc. (http://www.realnetworks.com) of Seattle, WA is a leader in media delivery over the Internet. RealNetworks offer several products for distribution of multimedia. RealNetworks also provides specialized, customizable developer tools for generic stream delivery. A drawback with RealNetworks solution is the requirement that custom software must be installed on the client side, namely, a specialized media server. Further, the RealNetworks products operate over a custom protocol - the Real-Time Streaming Protocol (RTSP).

Marimba Inc. (http://www.marimba.com) of Mountain View, CA is a leading provider of Internet solutions for automated deployment of applications and content. Such technology is commonly known as "Push Technology". In Push Technology transactions are typically initiated at the server based on individual user information. Push Technology provides management features and content replication. However, because transactions are initiated at the server, Push Technology would be difficult to adapt to real-time data delivery initiated by the client and would require significant software installation on the client side and a large server side infrastructure.

JFile from Software Artisans Inc. (http://www.softartisans.com) of Brookline, MA is a product that provides software-managed upload and retrieval of documents over the World Wide Web using a signed Java applet.

Microsoft Remote Scripting (http://msdn.microsoft.com/scripting/remotescripting/default.htm) from Microsoft Corporation of Redmond, WA is a technology that allows browser side (client) scripts to invoke server side scripts using HTTP as the transport protocol and XML as the

marshalling language. This technology is well suited for retrieving a small number of discrete items.

Ideally real-time transfer of data over the World Wide Web should be accomplished using standard protocols such as HTTP. Because it is cumbersome in decentralized organizations with many geographically dispersed locations to ensure that each such location has custom software available, to obtain a maximum benefit of using these technologies, it is desirable to minimize or eliminate the need for custom software at the client site.

From the foregoing it will be apparent that there is still a need to build on modern data transmission technologies such as the World Wide Web and the popular HTTP protocol to allow for real-time data streaming download from a web server to a browser using standard protocols and browser technology. It would be further desirable to provide a mechanism launch real-time applications at the client in conjunction with transfer of data in real-time using the HTTP protocol.

Summary of the Invention

In a preferred embodiment, the invention provides a mechanism for downloading files in real-time using the HTTP protocol without requiring extensive customization on the client side. In a system embodying the invention, the client side is capable of properly launching streaming applications and client side functionality is readily extended.

In one aspect, the invention may be embodied in a system for near real-time transfer of a datafile from a first computer to a second computer. Such a system has a first and a second computer both having a connection to a computer network and operable to communicate over the computer network using a standard protocol. On the server side computer a server side script, responsive to a download request from a second computer, is operable to launch an httpstreamproducer and to read and write data over the computer network using the standard protocol. The httpstreamproducer

reads a designated source file and simultaneously writes data from the source file into a return-data-buffer connected to the server-side script. A read-while-write mechanism allows the httpstreamproducer to read data from the designated source file while the designated source file is being written by a data producer program.

The second computer has a transaction handler class, each instance of which is operable to read and write data over the computer network using the standard protocol and to write blocks of data to a destination simultaneously with receiving data from the computer network.

The first computer may also have a webserver for transmitting a webpage containing a list of files available for download by other computers in which case the second computer has a corresponding webbrowser for displaying the webpage containing the list of files available for download.

The second computer may also have a trusted applet operable, in response to a user selecting a file from the list, to create a transaction handler instance for receiving the selected file. The second computer may also include at least one stream handler class having at least one file interaction method for performing a file operation selected from the set creating a file, opening a file and writing to a file, wherein the transaction handler instance creates a stream handler instance appropriate for the file selected by the user.

The standard protocol may for example be http or WAP.

The first computer may execute a webserver for transmitting a webpage containing a list of files for download by other computers and the second computer, a webbrowser for displaying the webpage containing the list of files available for download. The second computer may also execute a trusted applet which, in response to a user selecting a file from the list creates a transaction controller instance operable to manage a plurality of file transfer threads. Each file transfer thread, in response to the request from a user to download a file, executes a transaction controller instance to create a transaction handler instance for receiving data from the first computer.

In the second computer, a stream handler class has a method for receiving data from the transaction handler instance and for writing data to a destination. The destination may be a data file, an application program that is a data consumer, or a database.

In another aspect, the invention may be a method for near real-time download of a file via a computer network. According to that aspect the download of a file is accomplished by operating a client to select a file for download from a server, establishing a network link between a first process executing on the client and a second process executing on the server; reading at the server the selected file one block of data at a time; transmitting the block of data as a continuous stream on the link from the server to the client; and at the client, receiving the data as a continuous stream from the link and writing the data to a destination file one block at a time simultaneously to receiving the data.

One link may be shared between multiple stream producer/stream handler pairs. If that is the case, the data stream is broken up into data chunks each corresponding to one stream producer/stream handler pair.

In another aspect the invention may be an article of manufacture, namely, a program storage medium having computer readable program code means embodied therein, wherein the computer readable program code comprises instructions giving direction to a computer system, having a server side computer, a client side computer, and a computer network connecting the server side computer to the client side computer. These instructions cause the computer system to produce a list of files available for download from the server side computer and to display the list of files available for download on the client side computer. Further the instructions cause the computer system to allow a user to select on or more of the files available for download. In response to the selection of a file from the list, the computer readable instructions direct the computer system to create a transaction handler instance, wherein each transaction handler is operable to read and write data over the network and to transmit a request over computer network indicating to the server to transmit

the selected file. Further instruction include instructions to receive the request at the server and in response to receiving the request at the server read blocks of data from the selected file, place blocks of data in a return buffer, and to transmit the blocks of data from the return buffer to the client concurrently with reading additional blocks of data. Further instructions include instructions to receive the blocks of data at the client; and to write the blocks of data to a destination concurrently with receiving additional blocks of data.

In an alternative program medium aspect of the invention, the instructions include a web page producer, a web page reader, wherein the web page reader is operable to receive and to display a web page from the web page producer, a server side script operable to receive a download request and to launch an httpstreamproducer and to receive and transmit data over a standard protocol. The instructions also include an httpstreamproducer class each instance of which being operable to read a designated source file and simultaneously write data from the source file to a return-data-buffer; and a read-while-write mechanism providing the computer system instructions to enable the simultaneous reading from and writing to a data source. The server script causes the computer system to read data blocks from the return-data-buffer and to transmit the data blocks over the computer network. A transaction controller causes the computer system to receive a create instruction and in response to the create instruction, to create a transaction handler. The transaction handler is computer readable instruction that operate to cause the computer system to create an httpstreamhandler, to transmit get commands to a server side script, to receive blocks of data from the server side script; and to transfer the data to the httpstreamhandler. The httpstreamhandler is computer readable instructions to receive data from the transactionhandler; and to write data to a destination.

Other aspects and advantages of the present invention will become apparent from the following detailed description, taken in conjunction with the accompanying drawings, illustrating by way of example the principles of the invention.

Brief Description of the Drawings

Figure 1 is a system architecture diagram of a data delivery system embodying the invention.

Figure 2 is a data flow diagram illustrating the operation of an embodiment of the invention.

Figure 3 is an exemplary illustration of a web page listing files available for real-time download using an embodiment of the invention.

Figure 4 is a block diagram illustrating the architecture for a system for near real-time download according to the invention.

Detailed Description of the Preferred Embodiments

As shown in the drawings for purposes of illustration, the invention is embodied in a novel data delivery mechanism that has the ability to transfer file data, from a server to a client, in real-time over HTTP and to launch real-time applications. Various techniques for data transfer using HTTP exist. However, these techniques require extensive custom software at the client and cannot readily extend the client side functionality.

The present invention provides a generic mechanism for downloading data over in near real-time. In a preferred embodiment, the invention uses the HTTP and HTTPS protocols to transfer data between a server and a client. The invention provides for extensibility hitherto not achievable.

By way of example, the invention is described herein in the context of a data delivery system for the Petroleum E & P industry in which data is transferred from a data acquisition site (e.g., an oil field being explored) to a data delivery site (e.g., an oil company headquarters). The invention is equally applicable to other data delivery scenarios which may benefit from the real-time delivery of data. One example is the management and exploration for underground water resources. Another example is

the delivery of medical imaging data from a remote clinic to a hospital thereby allowing an expert physician aid a local physician in diagnosis and treatment of a patient. A third example is the delivery of video and sound images.

Furthermore, for exemplary purposes the invention is described below using the TCP/IP and HTTP data transmission protocols. The novel techniques described herein may be applicable to other existing and future protocols, for example, WAP.

Fig. 1 is a system architecture diagram of the data delivery system 10. The data delivery system with its framework components has been designed around the data to be handled, the data workflow, the time domains to be accommodated, and the variety of computer platforms and network connections available. Specifically, it has been designed around three main sites or functions: the acquisition site (wellsite) 11, the delivery site (operators' office) 12, and the auxiliary sites such as the data services center 13, data management center 14, and product delivery center 15.

These sites communicate through a secure central data hub 16. Although not explicitly shown in Fig. 1, there may be multiple delivery sites, auxiliary sites and acquisition sites connected to the central data hub 16. The hub 16 receives data and forwards it to the required locations, either to the delivery site 12, to an auxiliary site 13-15 or to the acquisition site 11. Real-time data delivery to the delivery site (in this case the operator's desktop) 12 may be achieved through the use of the HTTP protocol through a web data server 18 as described below in conjunction with Figure 2. The web data server 18 may be located either within a secure Intranet or within an associated secure enclave. The system can also accommodate point-to-point communication 17 directly between the acquisition site 11 and the delivery site 12.

Associated with this central data hub may be at least one product delivery center 15 comprised of specialized hardware and software systems designed specifically to generate hardcopy output in the form of products such as prints, tapes, films and CDs. The product delivery centers 15 may be located local to or in the operators' offices at the delivery site 12 or may be located virtually anywhere, removing the need for products to be generated at the acquisition site. Network

transmission to the local product delivery centers 15 greatly reduces product delivery times from remote acquisition sites. The central data hub 16, product delivery center 15 and/or web data server of choice 18 are typically, but are not required to be, co-located within a single data service center. The data delivery framework is flexible and can be configured in a number of ways. There are many permutations on the data delivery theme depending upon the preferences of an operator at project time, as well as the communications configuration of a given acquisition site.

Desktop hardware and software tools located on the operator desktop at the delivery site 12 or on desktops at the data services center 13 complete the data delivery framework system components. The tools facilitate the reception, handling and manipulation of data, received either physically or electronically, and assist the operators with their next step decision process, be that data integration, interpretation, processing or archiving.

Data delivery from the acquisition site 11, including both measurement data and job status information, may be transmitted over satellite, landline, microwave, ISDN, cell phone, direct Ethernet connection or by any method that supports the TCP/IP protocol or any other protocol that supports HTTP. Generally, either the operator or the service company provides communications from the well site. In either case, the service company's data acquisition system must include hardware and software to allow it to communicate over any of these various links using standard protocols. Since data files can be written over hours (wireline) or days (for, example, in logging-while-drilling (LWD) operations), the ability to transmit files as they are being created is an essential facet, crucial to timely decision-making.

A router-based mobile connection solution, designed to facilitate connection of the acquisition unit to the most common communications methods encountered ('standard modem' dial-up, ISDN or Ethernet) may be used. Intended for mobile systems that must reconfigure their network connection on a regular basis, it consists of a router, power supplies and connectors, along with a software interface preconfigured and ready to enable any Internet Protocol (IP) based network

application. It is designed for users who are not networking specialists and is straightforward to set up and run. The software 'manager' provides network and connectivity information and assists with troubleshooting, automatically indicating where and when a link has dropped out.

The data delivery system needs to transfer data from the often-remote temporary acquisition site 11 to a site hooked to an established communication infrastructure. The data delivery system uses, for example, the HTTP protocol as described below in conjunction with Figure 2.

The data delivery system 10 provides for interactive, real-time, collaborative viewing of acquisition site data in the operator's office 12, which is a key and growing need in today's E&P industry. This is especially true relative to interpreting critical drilling and logging data, both of which are used for 'next step' formation evaluation and well construction decision-making.

Specifically, drilling mechanics, resistivity and sonic data are delivered in real-time to facilitate pore pressure analysis for selecting casing points and minimizing fluid loss while drilling. Sonic (Delta-T) data while drilling are delivered to data service centers for integration and correlation with seismic data in order to "put the bit on the seismic map" and update the well plan in real time. LWD data are delivered for real-time integration into a reservoir model for the purpose of geosteering.

Getting the logging information to the right people at the right time and place—wherever they may be relative to the well site—may be achieved through point to point communications 17 using an interactive remote witness software package, originally designed for point-to-point (standalone), two way transmission.

These established real-time services comprise just one facet of the data delivery framework. Real-time communication allows specialists to provide timely expertise on multiple wells worldwide from a central location or multiple locations. Remote witnessing not only provides optimal use of key staff, but also reduces travel costs and personnel exposure to hazardous environments. Further to this, it facilitates

capture and dissemination of best practices, with the same staff collaborating on many wells in a specific field or region. Today's model for decision-making is thus becoming expert-centered versus asset-centered, including web-based real time remote witnessing.

Figure 2 is a data flow diagram illustrating the real-time bulk data transfer according to the invention. The data flow diagram of Figure 2 illustrates the transfer of a source file 201 residing on a server 211 to a client 213 where it may be stored as a destination file 203 or provided in real-time to a real-time application 205. The client side 211 may, for example, be the acquisition system 11 or the operator desktop 12 of Figure 1. The server 213 may be the web data server 18 of Figure 1.

In an embodiment of the invention, a data producer running on the server side produces data that is consumed by a data handler running on the client side. As an example, the data producer may be an HTTPStreamProducer 231 that reads from a data file 201. The corresponding data handler is a HTTPStreamHandler 229 running on the client side 213. The HTTPStreamProducer 231 and HttpStreamHandler 229 provide specific defined interfaces between the data transfer mechanism of the invention and the source and destination files.

The server side 211 and client side 213 are interfaced through a network 205. A user on the client side 213 interfaces with the system using a standard web browser such as Netscape Navigator or Microsoft Internet Explorer. Alternatively, the user uses a customized web browser that provides application specific functionality. In a preferred embodiment, the client side 213 functionality is provided by a web browser extended by a trusted Java applet 221, described herein below. The client-side functionality may also be implemented as a component that may be used by other application programs. An example of such an application are well-log interpretation and reservoir modeling systems, e.g., the Geoframe system from GeoQuest Corporation, Houston, Texas. In that embodiment of the invention, the application program would download a library component implementing the invention described herein without requiring the user to download or invoke a web browser.

In a preferred embodiment, a system according to the invention operates according to a pull-model. That is to say, a user at a client-side 213 initiates a data transfer from the server-side 211.

In a first step 1 the user browses to a DHTML web page 217 generated by a web server 215 on the server side 211 that displays links to source files available for download. Figure 3 is a screen shot of an exemplary web page 301. Having browsed to the web page 217 the web page is transferred in a standard manner to the 213 where it is displayed 219 to the user. The web page 301 contains a list of files 303a, 303b, and 303c available for download.

In a second step 2 the user, still interacting through the web browser selects a file from the list of available source files. Typically the user would select the file by clicking on a link associated with the file. In the preferred embodiment, the selection of the file activates a trusted applet 221. A trusted applet is a Java applet with a cryptographic signature applied to it so the identity of the author is certified. The signature, along with special software code, allows the applet to perform privileged operations such as establishing network connections or writing to files, which are generally not allowed by the security system in the Java runtime (also known as the "sandbox"). To obtain the higher privileges the trusted applet 221 asks for those privileges using browser specific APIs for that purpose. The trusted applet 221 may have been previously loaded. If the trusted applet 221 does not yet reside on the client side 213, it is automatically downloaded from the server side 211. The trusted applet 221 has an entry point, the get() method. The get() method is an implementation of a signature (i.e., function name and arguments) agreed-upon by the trusted applet 221 and the DHTML code of the web page 217.

The browser at the client side 213 invokes and passes the URL (Universal Resource Locator) string representing the remote source file 201 to the get() method. The URL points to a server-side script 223 (in a Microsoft implementation, the server-side script is an Active Server Pages script). The arguments for the get() message are specified at the end of the URL in the standard HTTP GET syntax. The HTTP GET

command is an HTTP command used by a client to request a server to return some data, e.g., a file. An example of the URL is:

http://ehub.com/downtest.asp?Handler=SerialFileHandler&Producer=SerialFileProducer&LocalName=file.pds

In the URL the arguments begin with "?" and are delimited by "&".

The next step, step 3, is to create a TransactionController instance. In the preferred embodiment, the get() method of the trusted applet 211 operates to create a TransactionController instance 225 in the thread that the get() method is executing in. The transaction controller 225 manages the worker threads that carry out the stream transfers. The transaction controller 225 creates new threads when the applet get() method is invoked, it forwards applet events (i.e., page transitions and applet shutdown) to the active threads, and shuts down the active threads when the browser exits.

In step 4, the TransactionController instance 225 creates a TransactionHandler thread 227 for the file to be downloaded. The TransactionHandler establishes a connection to a remote stream producer and moving data from the server-side ASP script 223 to a client-side HTTPStreamHandler instance 229. The HTTPSTreamHandler implements an open() method which when invoked creates a destination file.

In step 5, the TransactionHandler 227 creates the HTTPStreamHandler instance 229. If time-outs are enabled and no data is available the connection is timed out to prevent having open connections without activity. If the connection is timed-out, it is reestablished after a pre-defined waiting period. After this time-out management, the TransactionHandler 227 invokes the open() method of the HTTPStreamHandler 229. The open() method creates the destination file 203 and optionally launches the real-time application. The original URL string is passed to the open() call. The HTTPStreamHandler 229 modifies the arguments in the string as may be appropriate. The URL string is how the HTTPStreamHandler communicates with its server counterpart, the HTTPStreamProducer. In a preferred embodiment, a

real-time file transfer system according to the invention provides an error-recovery mechanism. If a part of a file is already present on the client-side 213 the HTTPStreamHandler 229 indicates in the URL string how much of the file is present to the server-side 211.

Alternatively, the received data may be directed to a real-time application, for example, a data viewer such as Schlumberger's PDSView program. That scenario is illustrated in Figure 2 using the TransactionHandler 227' and the HTTPStreamHandler 229'. If the data is directed to a real-time application, in addition to opening a destination file, the HTTPStreamHandler 229' launches the real-time application 205, step 5'. In an alternative embodiment, no destination file is opened and the data is directly streamed to the real-time application 205. A read while write mechanism 206 allows data to be written to a destination file 203' simultaneously as being presented to the real-time application 205.

In the discussion here in, for purposes of illustration, two stream handlers are shown: HTTPStreamHandler 229 and HTTPStreamHandler 229'. In practice there is no limit on how many stream handlers operate in parallel.

In step 6, the TransactionHandler 227 attempts to connect to the server-side 211 by sending an HTTP GET request using the URL string (possibly modified, if appropriate). Over a successfully established connection, the TransactionHandler 227 (or TranactionHandler 227') enters a state of being capable of receiving data from the server-side 211 via an https RESPONSE message. An HTTP GET message is a request from a client for a delivery of something (e.g., a file) specified in the argument presented to the HTTP GET. The HTTP RESPONSE message is the server's answer to the HTTP GET.

In the discussion here in, for purposes of illustration, two stream producers are shown: HTTPStreamProducer 229 and HTTPStreamProducer 229'. In practice there is no limit on how many stream producers operate in parallel. The HTTP protocol limits the number of connections between a client and a server to two. In one embodiment of the invention the stream of data from the server to the client may

service multiple HTTP stream producer - HTTP streamhandler pairs by breaking up the stream into multiple request-response pairs, wherein each request-response pair corresponds to a portion of a file to be downloaded. In that scenario the transaction handlers 227 alternate in accessing the data stream in a round-robin fashion.

In step 7, when the connection has been established, in response to the get() message, the server-side script 223 creates an appropriate type of HttpStreamProducer 231. An HttpStreamProducer 231 and a HttpStreamhandler 229 work together, that is to say, these components agree on the structure and meaning of the data stream. For example, an HttpStreamProducer 231 that reads data from a database should be paired with an HttpStreamHandler 229 that is designed to interpret the database stream. The ASP script 223 parses the URL string and creates the right HttpStreamProducer 231 based on the name provided.

An HttpStreamProducer is a server-side component that implements the producer interface (a preferred embodiment producer interface is set forth in the code appendix). The producer interface defines how a stream producing agent provides services to the server-side script 233. This common interface allows any agent to be used without regard to how it is implemented. Thus, you could have a database stream producer, and a serial file stream producer, and either could be accessed by a single ASP script via the common interface. The ASP script 223 calls the open() method of the HttpStreamProducer 231 to open the source file to be transferred.

The source file may be a source file that is in the process of being generated, e.g., from a data source 233. A data stream is fed from the data source 233 to a read while write mechanism 235. The data may then be simultaneously written to a source file 201' and transmitted to an HttpStreamProducer 231'.

For illustrative purposes two source files are shown: source file 201 and 201'. In practice many more files may exist or be in the process of being created on the server side 211.

The server-side script 223 calls the open() method of the HttpStreamProducer 231 or 231' passing it the URL string as an argument. If the call succeeds, the

SERVER-SIDE script 223 then repeatedly calls the getHeaderAt() method of the HttpStreamProducer to get any headers that should be passed to the client side 213 and adds these to the response message.

In step 8, to retrieve the data of the data file 201 or 201', the server-side script 233 repeatedly calls the fillBuffer() method of the HttpStreamProducer 231 or 231'. Each call to fillBuffer() prompts the HttpStreamProducer 231 or 231' to fill a buffer of data.

In step 9, the buffer of data that is returned from the call to fillBuffer() is written in a response message from the HttpStreamProducer 231 or 231' to the server-side script 223. In a preferred embodiment, a system according to the present invention provides for real-time transfer of data from a serial data file. The code appendix includes an implementation of the HttpStreamProducer interface called SerialFileProducer. The SerialFileProducer implementation (i.e., one HttpStreamProducer instance 231) of the HttpStreamProducer interface operates to produce a data stream from real-time serial file 201' using a Read-While-Write mechanism 235. If the HttpStreamProducer 231' is a SerialFileProducer and the buffer represents bytes read from a file, e.g., source file 201', that is being uploaded to the server-side 211.

In step 10, the server-side Script 223 upon receiving buffers of data from the HttpStreamProducer 231 or 231' transmits the data buffer in an https response message to where the data buffer is received by the TransactionHandler 227. The response is streamed continuously to the client side 213 over an open https connection.

In step 11, the TransActionHandler 227 upon receiving data over the open https connection, calls the WriteBlock() method of the HttpStreamHandler 229 or 229'.

The TransActionHandler 227 and HttpStreamHandler 229 or 229' for one transaction runs in a separate thread. When the transaction has been complete, i.e., all

data associated with a file has been received and processed, the TransActionHandler 227 and HttpStreamHandler 229 shut themselves down.

The functionality of an embodiment of the invention is readily extended by adding a new HttpStreamProducer and a new HttpStreamHandler and plugging these components into the system. A user wishing to use the extension downloads the new HttpStreamHandler class from server-side 211. When a file is downloaded for use with the extension, the trusted applet 221 creates an instance of the new HttpStreamHandler. The transport of data between the server-side 211 and client-side 213 proceeds as described above.

In a preferred embodiment, the process described above in conjunction with Figure 3 may be repeated for multiple files in parallel. While one or more file transfers are in progress the user may again select one of the available files from the web page 219 for download. This action by the user triggers the invocation of another get() on the trusted applet 221. The trusted applet 221 then directs the transaction controller 225 to create another transactionhandler instance 227 which, in turn, creates another HttpStreamHandler instance appropriate for this file download.

These further transaction handler 227 and HttpStreamHandler 229 execute in new and separate threads from each other, the trusted applet, and the previously executing transaction handlers 227 and HttpStreamHandlers 229.

Similarly on the server side 211, when the server-side Script 233 receives a further request for an additional file download, the server-side Script 233 creates a new HttpStreamProducer 231 instance appropriate for that file.

In one embodiment the communication between corresponding HttpStreamProducer-HttpStreamHandler pairs is carried out on a dedicated http connection between the server-side 211 and client-side 213. In an alternative embodiment, a fixed maximum number of connections are established. If the number of file transfers that are being carried out in parallel exceeds that maximum number, the client Java Runtime causes the files to be transmitted on the established connections in a shared fashion, for example, in a round-robin scheme.

Figure 4 is a block diagram showing the architecture for a system for near real-time download according to one embodiment of the invention. A server-side computer 211 is connected to a client-side computer 213 via network 205. The sever-side computer has a central processing unit (CPU) 401. Similarly, the client-side computer 213 has a central processing unit (CPU) 403. The server side CPU 401 is connected to one or more disk drives or other permanent storage system 405. For illustrative purposes, only one disk drive 405 is shown. A client-side computer 211 may have many disk drives or other permanent storage systems. The disk drive or storage system 405 stores the source files 201. Furthermore, the disk drive or storage system 405 stores the server-side script 223 and an HttpStreamProducer class 431.

To execute the method of the invention, for example, as described in conjunction with Figure 2, the CPU 401 loads the server-side script 223. Appendix A contains an exemplary server-side script 223. As discussed above in conjunction with the discussion of Step 7 of Figure 3, the server-side script creates an HTTPStreamProducer instance 231. That HTTPStreamProducer instance is derived from an HTTPStreamProducer class 431 stored on disk drive 405. Appendix B contains a program listing of an exemplary HTTPStreamProducer class 431.

Similarly, the description above in conjunction with Figure 2 describes the client-side creation of Transaction Controller instance 225, TransactionHandler instance 227, HttpStreamHandler instance 229. These are derived from the Transaction Controller class 425, the TransactionHandler class 427, and the HttpStreamHandler class 429. These classes are stored on disk drive 407 which is connected to CPU 403. Furthermore, the Trusted Applet 221 is also stored on disk drive 407.

In one embodiment, the data stream is compressed. The HttpStreamProducer 231 and HttpStreamHandler 229 are directed to turn on compression through the URL passed via the HTTP GET (Figure 2, step 6) and HTTP RESPONSE (Figure 2, step 10) commands, respectively. The compression may, for example, be the compression algorithm provided through the standard JAVA runtime environment. Other

compression algorithms may also be used. When compression is turned on, the HttpStreamProducer 231 is requested to provide a buffer of data through the fillbuffer() message (Figure 2, step 8), it compresses the data placed in the buffer before providing the data in the ReturnBuffer() message (Figure 2, step 9). The HttpStreamHandler 229, in turn, decompresses the data before writing the data to a destination file 203 or providing it to a real-time application 205.

One embodiment of the invention is implemented in the source code of the source code appendices, namely:

Appendix A - ClientApplet.java

Appendix B - Filedownload.asp

Appendix C - HttpStreamHandler.java

Appendix D - SerialFileHandler.java

Appendix E - SerialFileProducer.java

Appendix F - TransactionController.java

Appendix G - TransactionHandler.java

Although a specific embodiment of the invention has been described and illustrated, the invention is not to be limited to the specific forms or arrangements of parts so described and illustrated. The invention is limited only by the claims.

Appendix A

```
/* ClientApplet.java
 * Author: John Abney
 * Created: 7/12/99
 * Last Modified: 8/20/99
 *
 * This is the code for the applet itself.  Its main function is to
order the
 * TransactionController to spawn a new TransactionHandler.
 *
 */


import java.applet.*;

import netscape.security.PrivilegeManager;
import com.ms.security.PolicyEngine;
import com.ms.security.PermissionID;
import java.net.URL;
import java.net.URLConnection;
import java.net.HttpURLConnection;
import java.io.InputStream;
import java.io.IOException;

public class ClientApplet extends Applet {

      private TransactionController controller;
      protected static TrxChecker m_trx = null;
    protected static boolean m_trxCheckStarted = false;

    static
    {
        final String MSIE = "MSIE ";
        final String MSIE2 = "MSIE 2";
        final String MSIE3 = "MSIE 3";
        String  sUserAgent;

        try
        {
                if (Sysinfo.isWin32())
            {
                // we are Windows. Now make sure we are on IE 4 or
                // later by looking at the user agent
                    PolicyEngine.assertPermission(PermissionID.SYSTEM);
                sUserAgent = System.getProperty("http.agent");

                if ((sUserAgent != null) &&
                    (sUserAgent.indexOf(MSIE) != -1) && // it's IE
                    (sUserAgent.indexOf(MSIE2) == -1) && // it's not IE
2
                    (sUserAgent.indexOf(MSIE3) == -1)) // it's not IE 3
                {
                    m_trx = new TrxChecker();
                }
            }
        }
        catch (Exception e)
        {
```

```
    }
}

public String[][] getParameterInfo()
{
    String pinfo[][] = {
        {"trx_host", "String", "IP location of TRX server"}
    };

    return pinfo;
}

  public void init() {

        controller = new TransactionController();
  }

public void start() {

        controller.signalStart();

    // if using IE on Windows (indicated by a non-null trx checker
    // reference) test to see if we have access to the TRX server
        if (m_trx != null)
    {
        // get privileges so we can connect to TRX server
        // (if using IE on Windows)
            PolicyEngine.assertPermission(PermissionID.SYSTEM);

        Debug.TRACE("Browser can host TRX client");
        // if we have not started global thread to check for TRX
        // connectivity, do so now
          synchronized (m_trx)
        {
            if (!m_trxCheckStarted)
            {
                // the server location can be a parameter. If
                // the parameter is not there use the URL host
                String TrxHost = getParameter("trx_host");

                if (TrxHost == null || TrxHost.length() == 0)
                {
                    TrxHost = getDocumentBase().getHost();
                }

                m_trx.setHost(TrxHost);
                m_trx.start();
                m_trxCheckStarted = true;
            }
        }
    }

  }

  public void stop() {

        controller.signalStop();

  }

  public void destroy() {
```

```
            controller.signalDestroy();

      }

      public void get(String urlString) {

            KillableFrame fakeFrame = new KillableFrame("False frame");
            ModalOKDialog mokd;

            try {

                  // If the browser doesn't use either security model,
we have to bail out.

                  if ( (!Sysinfo.usesIESecurity()) &&
(!Sysinfo.usesNetscapeSecurity()) ) {

                        mokd = new ModalOKDialog(fakeFrame,
"Nonstandard Browser", "You'll need to use either Netscape Communicator
or Internet Explorer");
                        mokd.waitForUserResponse();
                        return;

                  }

                  // If the controller failed to be created, we can't
continue.

                  if (controller == null) {
                        mokd = new ModalOKDialog(fakeFrame, "Init
Error", "Applet failed to initialize correctly");
                        mokd.show();
                        return;
                  }

                  controller.startTransaction(urlString);

            }

            catch (SecurityException e) {

                  mokd = new ModalOKDialog(fakeFrame, "Permission
Needed", "You'll need to accept the certificate in order to run this
applet." + e.toString());
                  mokd.show();

            }

            catch (Exception e) {

                  mokd = new ModalOKDialog(fakeFrame, "Unexpected
Error", "An unexpected exception was received: " + e.toString());
                  mokd.show();

            }

      }

    public int getTrxStatus()
    {
```

```
        if (m_trx != null)
            return m_trx.getStatus();
        else
            return TrxChecker.STATUS_FAILED;
    }

    public int getTrxTestDuration()
    {
        if (m_trx != null)
            return m_trx.getElapsed();
        else
            return 0;
    }
}
```

Appendix B

```
<%
      option explicit

      Server.ScriptTimeout = 99999
      Response.Buffer = false

      dim sProducer, fio
      dim URL, Params, ReconstructedURL
      dim buffer
      dim headerName
      dim headerValue
      dim nextHeader
      Dim i
      dim lg

      sProducer = Request.QueryString ("Producer")
      Set fio = Server.CreateObject(sProducer)
    set lg = Server.CreateObject("EhubObjects.Log")

    lg.Facility = "FileDownld"
    URL = Request.ServerVariables("URL")
    Params = Request.ServerVariables("QUERY_STRING")
    ReconstructedURL = URL + "?" + Params

      lg.Log 0, "Opening stream with " + sProducer, 0
    on error resume next
    fio.open ReconstructedURL

      if err <> 0 then
        lg.Log 3, "Producer open failure: " & Err.description, 0
        Response.Status = "404 Not Found"
        Response.AddHeader "HttpStreamProducerError", Err.description
        Response.End
      end if

      nextHeader = 1
      fio.getHeaderAt nextHeader, headerName, headerValue
      while err = 0
        Response.AddHeader headerName, headerValue
        nextHeader = nextHeader + 1
        fio.getHeaderAt nextHeader, headerName, headerValue
      wend

    Err.clear

    fio.fillBuffer 65536, buffer

    do while err = 0
        ' only write non-empty buffers back to the client. it was found
        ' that empty buffers were actually interpreted as data by
        ' Netscape, producing file corruption.
        if(Ubound(buffer) > -1) then
            response.binarywrite buffer
        end if
        fio.fillBuffer 65536, buffer
        if not Response.IsClientConnected then
           lg.Log 2, "Client disconnected", 0
```

```
            exit do
        end if
    loop

    fio.Close
%>
```

Appendix C

```
/* HttpStreamHandler.java
 * Author: John Abney
 * Created: May 26, 1999
 * Last modified: 8/20/99
 *
 * This is the interface defining client-side objects that will handle
the task of
 * "handling" data as it is received from the server-side
HttpStreamProducer.
 *
 */

import java.io.IOException;
import java.net.MalformedURLException;
import java.util.Dictionary;

public interface HttpStreamHandler extends EventReceiver {
      public void setEventReceiver(EventReceiver eventReceiver);
      public String open(String urlString) throws IOException,
MalformedURLException, ClientNotInterestedException;
      public void writeBlock(byte[] dataToWrite, int size) throws
IOException, ClientNotInterestedException;
      public void reportHeaders(Dictionary headers) throws
ClientNotInterestedException;
      public long getFlags();
      public void close() throws IOException;

}
```

Appendix D

```
/* SerialFileHandler.java
 * Author: John Abney
 * Created: 5/26/99
 * Last modified: 8/20/99
 *
The SerialFileHandler is one implementation of the
HttpStreamHandler.  It is the
 * relatively straightforward case of the eHub server transporting a
file to the client.
 *
 */

import java.io.*;
import java.awt.*;
import java.util.zip.*;
import java.util.Dictionary;
import java.net.MalformedURLException;

public class SerialFileHandler implements HttpStreamHandler {

      private static final String SEMAPHORE_MAGIC = "RWWv1.0\n";  //
Must include newline!


      /* Static variables */
      private static String defaultDestDirectory = null;


      /* Static initializer */

      static {

            defaultDestDirectory = null;

      }


      // Public interface

      /* SerialFileHandler()
       * Initialize all non-static member variables to something safe.
       *
       */

      public SerialFileHandler() {

            Debug.TRACE("In SerialFileHandler()");

            openedSuccessfully = false;
            preserveSemaphore = false;
            keptAlive = false;
            signaledDataExists = false;
            signaledDataFinished = false;
            transferLength = -1;
            bytesWritten = 0;

            destinationFile = null;
```

```
            destinationStream = null;
            semaphoreFile = null;
            urlBuilder = null;
            parentFrame = null;
            headerDictionary = null;

        alteredURLString = null;

            esQueue = new EventSignalQueue();
            initFlags();

      }


      /* open()
       *
       * Must check for and attempt to open a local version of the
 file.
       * If no local version is found, then one is created along with a
       * semaphore file.  If a local version is found, then other
functions
       * are called to determine what should be done.  Once the class
has determined
       * what it must do, it will build and return a corrected (or
unchanged) urlString.
       *
       */

      public String open(String urlString)
      throws IOException, MalformedURLException,
ClientNotInterestedException {

            Debug.TRACE("open() called with " + urlString);

            if (keptAlive) {

                  String currentSize =
String.valueOf(destinationFile.length());
                  urlBuilder.removeParameter(URLParameterEnum.START_AT);
                  urlBuilder.addParameter(URLParameterEnum.START_AT,
currentSize+1);
                  keptAlive = false;
                  preserveSemaphore = false;
            alteredURLString = urlBuilder.getCurrentURLString();
                  return alteredURLString;

            }

            if (openedSuccessfully) {
                  Debug.TRACE("WARNING: The SerialFileHandler was
opened twice without being closed.");  // TODO: More subtle?
                  // Debug.ASSERT(false, "The SerialFileHandler was
opened twice without being closed.");
            }

            Debug.ASSERT(urlString != null);     // Make it clear when
this happens for debug
            if (urlString == null)                     // Still
serious, throw the exception
                  throw new MalformedURLException();
```

```
            urlBuilder = new CustomURLBuilder(urlString);

        try
        {
                while (alteredURLString == null)
                  alteredURLString = prepareHandler(urlString);   // URL
String will be appropriately changed
        }
        catch (ClientNotInterestedException cnie)
        {
            // make sure if we are bailing out, we don't remove an
            // existing semaphore
            preserveSemaphore = true;
            throw cnie;
        }
        catch (IOException ioe)
        {
            String Msg = "Error opening local output file " +
                         destinationFile.getAbsolutePath() + ": " +
                         ioe.getMessage();

            // probably we could not open the output file. Inform the
user and
            // raise a ClientNotInterestedException.
                ModalOKDialog mokd = new ModalOKDialog(parentFrame,
"File Output Error", Msg);
            mokd.show();
                throw new ClientNotInterestedException(Msg);
        }

            Debug.TRACE("open(): Returning " + alteredURLString);
            openedSuccessfully = true;

            Debug.ASSERT(alteredURLString != null);

            return alteredURLString;

      }


      /* reportHeaders()
       * The headers that were received when the connection was
restarted are
       * passed in through here in the form of a Dictionary object,
probably
       * a Hashtable.
       *
       */

     public void reportHeaders(Dictionary headers) throws
ClientNotInterestedException {

            headerDictionary = headers;

            processLengthHeaders();
            processErrorHeaders();

            parentFrame.startThroughputTimer();

      }
```

```
      /* writeBlock()
       * The idea is that writeBlock() will take in a byte[] and write
       * it out to the currently opened destinationStream.  However, if
       * destinationStream is null, that means that the caller forgot
to first
       * call the open() method, which they will regret doing, let me
tell you.  The
       * solution is that we throw an IOException back at them.
       *
       * The block is then written.  If an IOException occurs, this
means that there
       * was some general failure with the file writing mechanism and
very little can be
       * done.  If we get an ArrayOutOfBoundsException, then we know
that either the byte[]
       * we were passed was too small or the size that we were given to
use was too large.
       * We need to handle either case.
       *
       */

      public void writeBlock(byte[] dataToWrite, int size)
      throws IOException, ClientNotInterestedException {

            processWaitingSignals();

            if (checkCancelFlag()) {
                  preserveSemaphore = true;
                  throw new ClientNotInterestedException();
            }

            if (!openedSuccessfully) {
                  displayErrorMessage("A write was attempted to an
unopened FileStreamHandler");
                  return;
            }

            try {
                  destinationStream.write(dataToWrite, 0, size);

                  // signal data exists to launcher on the first
                  // arrival
                  if (!signaledDataExists)
                  {
                        try
                        {
                              launcher.signalDataExists();
                              signaledDataExists = true;
                        }
                        catch (FailedLaunchException fle)
                        {
                        }
                  }

                  if (compressionLevel >
CompressionEnumeration.NO_COMPRESSION) {
                        try { bytesWritten =
((DecompressOutputStream)destinationStream).numWritten(); }
                        catch(ClassCastException e) {
```

```
                              Debug.TRACE("WARNING!  writeBlock():
Compression settings are incorrect - " + e.toString());
                              bytesWritten += size;
                        }
                  }
                  else
                        bytesWritten += size;

                  // Otherwise, the compression stream will handle it

                  // Debug.TRACE("writeBlock() wrote " + size + " bytes
(total " + bytesWritten + ")");
                  parentFrame.setCount(bytesWritten);
            }
            catch (IOException e) {
                  Debug.TRACE("writeBlock(): Write failed");
                  throw e;
            }
            catch (ArrayIndexOutOfBoundsException e) {
                  if (size < 0)
                        displayErrorMessage("Negative size parameter in
writeBlock()");
                  else
                        displayErrorMessage("Size parameter too large in
writeBlock()");
                  Debug.ASSERT(false, "Bad size parameter in
writeBlock()");
            }

            return;
      }


      /* signal()
       * This method allows the TransactionHandler to send
EventSignal's to the
       * handler.  These signals may be ignored or acted upon,
depending on the
       * specific purpose of the handler.
       *
       * In this case we act upon the CONNECTION_BAD signal and
preserve all of
       * the others.
       *
       */

      public synchronized void signal(EventSignal sig) {

            Debug.TRACE("signal(): received " + sig.toString());
            int eventType = sig.getEventType();

            switch(eventType) {

            case EventSignal.CONNECTION_BAD:

                  keptAlive = true;
                  preserveSemaphore = true;
                  break;

            case EventSignal.CONNECTION_SLOW:
                  keptAlive = true;
```

```
                preserveSemaphore = true;
                break;

            case EventSignal.CONNECTION_NOT_ESTABLISHED:
                keptAlive = false;
                preserveSemaphore = false;

                // keep the launcher from running
                signaledDataExists = true;
                signaledDataFinished = true;
                break;

            }

        }


        /* getFlags()
         * This method returns a series of true/false flags that describe
the
         * characteristics of the FileStreamHandler.
         *
         */

        public long getFlags() {

            return flags;

        }



        /* close()
         * This method should clean up any "messes" that the class might
have made in its
         * lifetime.  These could include any streams that need to be
closed or files that
         * need to be deleted.  The obvious things are to
         *          a) close() the destinationStream
         * and  b) deleteSemaphoreFile()
         *
         * It's also important to be sure that we don't clean up any
resources that may not
         * have been allocated or we run the risk of eating a
NullPointerException.
         *
         */

        public void close() throws IOException {

            Debug.TRACE("in close()");
            try {
                if (!openedSuccessfully)
                    Debug.TRACE("close() called on unopened
SerialFileHandler");
                else if (keptAlive) {
                    destinationStream.flush();
                    Debug.TRACE("close(): keptAlive is true");
                }
                else {
                  Debug.TRACE("close - closing destination stream");
```

```
                    if (compressionLevel !=
CompressionEnumeration.NO_COMPRESSION)
                {
                            destinationStream.flush();
                            destinationStream.close();
                }
                        fileStream.close();
                if (launcher == null)
                    Debug.TRACE("oops, launcher is null");
                if (signaledDataFinished)
                    Debug.TRACE("we have already signaled, so we are
not launching");
                if (launcher != null && !signaledDataFinished)
                        {
                        Debug.TRACE("close - signaling launcher");
                    launcher.signalDataFinished();
                            signaledDataFinished = true;
                        }
                    }
            }

        catch(FailedLaunchException e) {
            Debug.TRACE("close - Caught FailedLaunchException: " +
e.getMessage());
            }

            catch(IOException e) {
                  Debug.TRACE("destinationStream failed to close()
properly in SerialFileHandler.close()");
                  throw e;
            }
        catch(Exception e)
        {
            Debug.TRACE("close - exception caught: " + e.getMessage());
        }
            finally {
                  if (!preserveSemaphore)
                        deleteSemaphoreFile();

                  if ((parentFrame != null) && (keptAlive == false)) {
                        parentFrame.setVisible(false);
                        parentFrame.dispose();
                  }

                  if (keptAlive == false) {
                        destinationStream = null;
                        destinationFile = null;
                        semaphoreFile = null;
                        urlBuilder = null;
                        openedSuccessfully = false;
                  }

            }

      }


      // Private methods


      /* initFlags()
```

```
        * This method sets the appropriate flags so that they may be
filled in when
        * the object is instantiated.  This method is not static, since
it's very
        * possible that down the road, these flags may vary not only
from class to class,
        * but even from instance to instance.
        *
        */

       private void initFlags() {

             flags = 0;
             flags |= HttpStreamHandlerFlagEnum.NEEDS_CLEANUP_TIME;
             flags |=
HttpStreamHandlerFlagEnum.DISCONNECT_RECONNECT_CAPABLE;

             Debug.TRACE("initFlags() set flags to: " + flags);

       }


       /* lookupHeader()
        * This method performs a convenient lookup in the
headerDictionary.
        *
        */

       private String lookupHeader(String key) {

             return (String)headerDictionary.get(key);

       }


       /* displayErrorMessage()
        * This function is used to display error messages to the user.
Eventually, it will
        * probably also consist of a dialog box, but for now we can just
TRACE and print to
        * System.out.
        *
        */

       private void displayErrorMessage(String errorMessage) {

             // TODO: Build in a dialog box for error msg displays
             Debug.TRACE("displayErrorMessage() got: " + errorMessage);

       }



       private void processLengthHeaders() {

             String lengthString =
lookupHeader(HeaderEnumeration.CONTENT_LENGTH);

             if (lengthString != null) {
                   transferLength = Integer.parseInt(lengthString);
```

```
                Debug.ASSERT(parentFrame != null);
                Debug.ASSERT(destinationFile != null);

                if (parentFrame != null) {
              parentFrame.setMaxCount(transferLength);
                }

          }

          else
                parentFrame.setVisible(true);


    }


    private void processErrorHeaders() throws
ClientNotInterestedException {

          String errorString =
lookupHeader(HeaderEnumeration.PRODUCER_ERROR);

          if (errorString == null)
                return;

          ModalOKDialog mokd = new ModalOKDialog(parentFrame, "File
not found", "Sorry, that file was not found on the remote server.");
          mokd.show();
          throw new ClientNotInterestedException("File not found on
remote server");

    }


    /* prepareHandler()
     * This method is called repeatedly in an attempt to finalize the
eventual
     * destination of the file.  If it fails to find a suitable
destination, it
     * should return null.  After prepareHandler() returns null, the
caller should
     * call prepareHandler again with the updated urlString.
     *
     */

    private String prepareHandler(String urlString)
    throws ClientNotInterestedException, MalformedURLException,
IOException {

          Debug.TRACE("In prepareHandler() with " + urlString);

          destinationFile = getDestinationFile(urlString);
          semaphoreFile = getSemaphoreFile(destinationFile);

          if (parentFrame == null) {
                Debug.TRACE("***Building parent frame!");
          parentFrame = new ProgressDialog(destinationFile.getName(),
                                          "Waiting to begin
transfer...",
                                          -1,
```

```
                                                this);
                        // parentFrame.show();
                        Debug.TRACE("Built parent frame");
                }

                Debug.TRACE("semaphoreFile = " +
semaphoreFile.getAbsolutePath());

                compressionLevel = getCompressionLevel();

                if (destinationFile.exists()) {

                        if (semaphoreFile.exists())
                                return handleHaltedTransfer(urlString);
                        else
                                return handleExistingLocalFile(urlString);

                } else {

                        if (semaphoreFile.exists())
                                return handleOrphanedSemaphore(urlString);
                        else
                                return handleNormalTransfer(urlString);

                }

        }


        /* getDestinationFile()
         * This method needs to determine the appropriate destination
file based on the
         * URL that it's passed as a parameter.  Note that it does not
yet create the file
         * if it doesn't exist.
         *
         */

        private File getDestinationFile(String urlString) throws
MalformedURLException, ClientNotInterestedException {

                // TODO: Verify the correctness of this approach, throw
better exception
                Debug.TRACE("In getDestinationFile()");

                String fileName =
urlBuilder.lookupParameter(URLParameterEnum.LOCAL_NAME);

                if (fileName == null)
                        throw new MalformedURLException();  // TODO: Correct
behavior?
                else
          {
                //Is it a silent mode?
                boolean bSilent = getSilentMode();

                if(bSilent == false)
                                return promptForLocalFile(fileName);
                else
                {
```

```
                //silent mode - assume that the file name is the
complete file name
                return new File(fileName);
            }

        }

            // Possibly verify that the file name is appropriate for
the system???

      }



      /* promptForLocalFile()
       * This method prompts the user to choose a local file.  It will
use the fileName
       * passed in as a default, but will also allow the user to choose
another name.
       *
       */

      private File promptForLocalFile(String defaultFileName) throws
ClientNotInterestedException {

            Debug.TRACE("In promptForLocalFile()");
            Frame f = new Frame();

            FileDialog fileSelect = new FileDialog(f, "Save File
Location");
            Debug.TRACE("Instantiated");
            String selectedFileName, selectedDirectory;


            fileSelect.setFile(defaultFileName);
            Debug.TRACE("setfile");
            if (defaultDestDirectory != null)
                  fileSelect.setDirectory(defaultDestDirectory);

            Debug.TRACE("About to show() selection dialog");
            fileSelect.show();
            Debug.TRACE("Selection dialog returned from show()");
            selectedFileName = fileSelect.getFile();
            selectedDirectory = fileSelect.getDirectory();
            fileSelect.dispose();

            // TODO: Is this the right way to handle a cancellation?...
            if ( (selectedFileName == null) || (selectedDirectory) ==
null )
                  throw new ClientNotInterestedException();

            Debug.ASSERT(selectedDirectory != null);
            Debug.ASSERT(selectedFileName != null);
            defaultDestDirectory = selectedDirectory;

            Debug.TRACE("promptForLocalFile(): got " +
selectedDirectory + selectedFileName);

            return new File(selectedDirectory + selectedFileName);

      }
```

```
      /* getSemaphoreFile()
       * This method creates a semaphore file based off of any give File
object.  It
       * does not write the file, however, and thus doesn't create it
if it doesn't
       * already exist.
       *
       */

      private File getSemaphoreFile(File baseFile) {

            // TODO: Don't forget special cases for other machines, do
that later
            Debug.TRACE("In getSemaphoreFile()");
            Debug.ASSERT(baseFile != null);
            return new File(baseFile.getAbsolutePath() + "_smf");

      }


      /* userClickedCancel()
       * This method is called by the ProgressDialog whenever the user
clicks the cancel
       * button.
       *
       */


      public synchronized void userClickedCancel() {

            cancelFlag = true;
            preserveSemaphore = true;
        if (transactionEventReceiver != null)
              transactionEventReceiver.signal(new
EventSignal(EventSignal.CLIENT_NOT_INTERESTED));
      }


      /*  checkCancelFlag()
       *  This method allows us to synchronously check the cancelFlag.
       *
       */

      private synchronized boolean checkCancelFlag() {

            return cancelFlag;

      }


      private int getCompressionLevel() {

            int theNum;

            try {
                  theNum =
Integer.parseInt(urlBuilder.lookupParameter(URLParameterEnum.COMPRESSIO
N_LEVEL));
            }
```

```
            catch(NumberFormatException e) {
                  return 0;
            }

            return theNum;

      }

    private boolean getSilentMode()
    {
        boolean bSilent = false;

          String sSilent =
urlBuilder.lookupParameter(URLParameterEnum.SILENT);
            if (sSilent != null)
        {
              bSilent = Boolean.valueOf(sSilent).booleanValue();
        }

        Debug.TRACE("getSilentMode() is " + bSilent);
        return bSilent;
    }

      /* processWaitingSignals()
       * This method reacts to all signals waiting in the esQueue.
Note that
       * all signals are currently ignored.
       *
       */

      private void processWaitingSignals() {

            EventSignal es;
            while (esQueue.signalsWaiting()) {

                  es = esQueue.getNextSignal();
                  if (es.getEventType() == EventSignal.CONNECTION_BAD)
{
                        Debug.ASSERT(false, "Should have been
handled!");
                        keptAlive = true;
                        preserveSemaphore = true;
                  }

            }

      }


      /* handleBadConnectionsignal()
       * This method alerts the user that a connection problem has
occured.
       * It then prompts the user in yes/no fashion whether or not they
would
       * like to continue the transfer.
       *
       */

      private boolean handleBadConnection() {
```

```
            BadConnectionDialog bcd = new
BadConnectionDialog(parentFrame, "Connection Problem",

              "A connection problem has occurred.  Would you like
attempt to reconnect?");
            return bcd.waitForUserResponse();

      }


      /* handleNormalTransfer(String urlString)
       * This method handles the case where the file that is about to
be downloaded does
       * not have a local version or semaphore file that it's colliding
with.
       *
       */

      private String handleNormalTransfer(String urlString) throws
IOException {

            Debug.TRACE("In handleNormalTransfer()");

            fileStream = new
FileOutputStream(destinationFile.getAbsolutePath());
            destinationStream = createDestStreamAndLauncher(fileStream,
urlString);

            writeSemaphoreFile();
            return urlString;       // Assumption: Remains the same if no
local version

      }


      /* handleHaltedTransfer()
       * Handles the case where an existing local file is found along
with its
       * associated semaphore.  The implication is that this transfer
has been tried before
       * and was interrupted, though this is not necessarily the case.
       *
       */

      private String handleHaltedTransfer(String urlString) throws
IOException, ClientNotInterestedException {

            // TODO: Prompt the user to explain what they'd like to do,
and
            // build the correct URL to carry out their request.

            Debug.TRACE("In handleHaltedTransfer()");
            Debug.ASSERT(destinationFile.exists());

            int userChoice = userRecoverySelection();

            // TODO: Case where the user wants to pick up where they
left off
            switch(userChoice) {

            case RecoverySelectionEnum.RECOVER:
```

```
                    Debug.TRACE("User chose to recover");
                    return handleRecovery(urlString);
                case RecoverySelectionEnum.OVERWRITE:
                    Debug.TRACE("User chose to overwrite");
                boolean deleted = destinationFile.delete() &&
                                  deleteSemaphoreFile();
                if (!deleted)
                {
                        Debug.TRACE("Could not delete output or semaphore
file");
                    ModalOKDialog mbox = new ModalOKDialog(parentFrame,
                                                           "Output File
Locked",
                                                           "Could not
overwrite " +
                                                           destinationFile.getName() +
                                                           ". The file is
locked by another application.");
                    mbox.show();
                    throw new ClientNotInterestedException("Output file is
locked");
                }
                else
                        return handleNormalTransfer(urlString);
                case RecoverySelectionEnum.NEWFILE:
                    Debug.TRACE("User chose to refer to a different
file");
                    return null;
                default:
                    Debug.ASSERT(false, "Fell through switch in
handleHaltedTransfer()");
                    return null;

                }

        }


        /* userRecoverySelection()
         * This method brings up a dialog box that asks the user whether
or not they would like to recover the old transfer.
         * It will give them three options: Recover, Overwrite, or
Cancel.  If they elect to Recover, we'll just start tacking
         * stuff onto the end of the file like any normal recovery.  If
they elect to Overwrite, we're gonna erase what they have
         * on there so far.  If they cancel, we'll just throw a
ClientNotInterestedException and close up shop.  And if they
         * decide that they want to pick a different file name or
directory location, we'll let them do that as well by return
         * null.
         *
         */

        private int userRecoverySelection() throws
ClientNotInterestedException {

            // TODO: Bring up a dialog box that prompts the user about
whether or not they wanna recover
            // Should we bring up an extra "WARNING" message?  Is that
babying the client too much?
```

```
            Debug.TRACE("In userRecoverySelection()");

            ExistingFileAndSemDialog rsd = new
ExistingFileAndSemDialog(parentFrame, "Recovery Dialog Box", "A file
with that name already exists and appears incomplete. " +

"Would you like to recover, overwrite, or select a different file? " +

"Before proceeding, make sure file is not already being downloaded.");

            int result = rsd.waitForUserResponse();

            if (result == RecoverySelectionEnum.CANCEL)
                  throw new ClientNotInterestedException();

            return result;

      }


      /* userExistingFileSelection()
       * This method is called if the user has selected a file that
already exists that does not
       * have an associated semaphore file to go along with it.
       *
       *
       */

      private int userExistingFileSelection() throws
ClientNotInterestedException {

            Debug.TRACE("In userExistingFileSelection()");

            ExistingFileDialog fcsd = new
ExistingFileDialog(parentFrame, "Recovery Dialog Box", "A file with
that name already exists and appears complete. " +

"Would you like to overwrite or select a different file? ");

            int result = fcsd.waitForUserResponse();

            if (result == RecoverySelectionEnum.CANCEL)
                  throw new ClientNotInterestedException();

            return result;

      }


      /* handleRecovery()
       * If the user elects to recover the destination file, we get it
ready for them here.
       *
       */

      private String handleRecovery(String urlString) throws
IOException, ClientNotInterestedException {

        RandomAccessFile    randFile;
        int                 fileLen;
```

```
        Debug.TRACE("In handleRecovery()");
            Debug.ASSERT(destinationFile.exists());

            bytesWritten = destinationFile.length();
        fileLen = (int)destinationFile.length();
            String currentSize = String.valueOf(fileLen + 1);
            urlBuilder.addParameter(URLParameterEnum.START_AT,
currentSize);  // Establish a standard

            if (!semaphoreFile.exists())
                  writeSemaphoreFile();

        // use a random access file to make sure we start at the right
spot,
        // in case there are concurrent writers.
        randFile = new
RandomAccessFile(destinationFile.getAbsolutePath(), "rw");
        randFile.seek(fileLen);
            fileStream = new FileOutputStream(randFile.getFD());

            String constructedURL = urlBuilder.getCurrentURLString();
            destinationStream = createDestStreamAndLauncher(fileStream,
constructedURL);

            Debug.TRACE("Returning " + constructedURL + " from
handleRecovery()");


            return constructedURL;
      }



      /* handleExistingLocalFile()
       * This method handles the case where an existing local file is
found but
       * without its associated semaphore.  The implication here is
that a previous
       * transfer of the file has succeeded or that another one of the
user's files
       * is in the way.
       *
       */


     private String handleExistingLocalFile(String urlString) throws
IOException, ClientNotInterestedException {

            // TODO: Prompt the user to explain what they'd like to do,
and
            // build the correct URL to carry out their request.

            Debug.TRACE("In handleExistingLocalFile()");
            Debug.ASSERT(destinationFile.exists());

        int userChoice;

        //in silent mode we override the file without prompting the
user.
        boolean bSilent = getSilentMode();
```

```
        if(bSilent == true)
        {
            Debug.TRACE("Silent Mode: Automatically overwriting
file.");
            userChoice = RecoverySelectionEnum.OVERWRITE;
        }
        else
        {
            //ask the user
            userChoice = userExistingFileSelection();
        }

            // TODO: Case where the user wants to pick up where they
left off
            switch(userChoice) {

        // NOTE: recovery is not supported if semaphore file is absent
            // case RecoverySelectionEnum.RECOVER:
            //    Debug.TRACE("User chose to recover");
            //    return handleRecovery(urlString);
            case RecoverySelectionEnum.OVERWRITE:
                  Debug.TRACE("User chose to overwrite");
            boolean deleted = destinationFile.delete() &&
                              deleteSemaphoreFile();
            if (!deleted)
            {
                      Debug.TRACE("Could not delete output or semaphore
file");
                ModalOKDialog mbox = new ModalOKDialog(parentFrame,
                                                       "Output File
Locked",
                                                       "Could not
overwrite " +
destinationFile.getName() +
                                                       ". The file is
locked by another application.");
                mbox.show();
                throw new ClientNotInterestedException("Output file is
locked");
            }
            else
                      return handleNormalTransfer(urlString);
            case RecoverySelectionEnum.NEWFILE:
                  Debug.TRACE("User chose to refer to a different
file");
                  return null;
            default:
                  Debug.ASSERT(false, "Fell through switch in
handleExistingLocalFile()");
                  return null;

            }

      }


      /* handleOrphanedSemaphore()
       * This method handles the case where an existing semaphore is
found, but no
       * associated local file is found.  This indicates that a
```

```
transfer was perhaps not
        * completed previously, and something happened to the local
file.
        *
        */

      private String handleOrphanedSemaphore(String urlString) throws
IOException {

            // TODO: What to do in this case?  For now, delete it and
forget about it.
            Debug.TRACE("In handleOrphanedSemaphore()");
            Debug.ASSERT(semaphoreFile.exists());
            Debug.ASSERT(!destinationFile.exists());

            semaphoreFile.delete();
            return handleNormalTransfer(urlString);

      }


      /* writeSemaphoreFile()
       * This writes out the semaphore, signalling the beginning of a
session.  If the
       * file already exists, displayErrorMessage() is called.
       *
       */

      private void writeSemaphoreFile() throws IOException {

            Debug.TRACE("In writeSemaphoreFile()");

            if (semaphoreFile.exists()) {
                  Debug.TRACE("Semaphore file already exists");
                  Debug.ASSERT(false, "Wrote existing semaphore file");
                  // necessary to throw an exception?
                  return;
            }

            FileOutputStream semFos = new
FileOutputStream(semaphoreFile);
            PrintWriter semPw = new PrintWriter(semFos);
            semPw.print(SEMAPHORE_MAGIC);
            semPw.flush();
            semPw.close();
            semFos.close();

      }


      /*  createDestinationStream()
       *  If we need to offer decompression, we return a deflating
       *  OutputStream.  Otherwise, we just return the normal stream.
       *
       */

      private OutputStream createDestStreamAndLauncher(FileOutputStream
fos,
                                                       String urlString) {

            LauncherFactory lf = new LauncherFactory();
```

```
            String launcherName =
urlBuilder.lookupParameter(URLParameterEnum.LAUNCHER);
            launcher = lf.makeLauncher(launcherName,
destinationFile.getAbsolutePath(),
                                       urlString);

            if (compressionLevel ==
CompressionEnumeration.NO_COMPRESSION) {
                  Debug.TRACE("createDestStreamAndLauncher(): created
uncompressed stream");
                  return fos;
            }
            else {
                  Debug.TRACE("createDestStreamAndLauncher(): created
compressed stream");
                  return new DecompressOutputStream(fos);
            }

      }


      /* deleteSemaphoreFile()
       * This deletes the semaphore, signalling the end of a session.
If the file does
       * not exist, we TRACE that, but it doesn't necessarily indicate
an internal error.  It
       * could be caused by the wrapper applet calling close() before
making a call to
       * open().
       *
       */


      private boolean deleteSemaphoreFile() {
            boolean deleted = true;

            Debug.TRACE("In deleteSemaphoreFile()");

            if (semaphoreFile == null) {
                  Debug.TRACE("Got null file in
deleteSemaphoreFile()");
                  return deleted;
            }

            if (!semaphoreFile.exists()) {
                  Debug.TRACE("deleteSemaphoreFile() called on
nonexistent file.");
                  return deleted;
            }

            deleted = semaphoreFile.delete();
            Debug.TRACE("deleteSemaphoreFile() returning");
            return deleted;
      }

    public void setEventReceiver(EventReceiver eventReceiver)
    {
        transactionEventReceiver = eventReceiver;
    }

      // Private member variables
```

```
        private ProgressDialog parentFrame;

        private File destinationFile;
        private OutputStream destinationStream;
        // private BufferedOutputStream bufferedStream;
        private FileOutputStream fileStream;

        private File semaphoreFile;
        private CustomURLBuilder urlBuilder;

        private Launcher launcher;
        private Dictionary headerDictionary;

        private int transferLength;
        private int compressionLevel;

        private EventSignalQueue esQueue;

        private boolean openedSuccessfully;
        private boolean preserveSemaphore;
        private boolean cancelFlag;
        private boolean keptAlive;
        private boolean signaledDataExists;
        private boolean signaledDataFinished;
        private long bytesWritten;

        private long flags;

        private String alteredURLString;
    private EventReceiver transactionEventReceiver;
}
```

Appendix E

```
/* SerialFileProducer.java
 * Author: John Abney
 * Created: 6/1/99 ??
 * Last Modified: 7/9/99
 *
 * This is the Java class representing the server-side producer of
serial files
 * for the client's consumption via the SerialFileHandler.  This java
class is
 * wrapped in a COM object so that it can be used through an ASP.
 *
 *
 */


import java.io.*;
import java.net.*;
import java.util.zip.*;
import java.util.NoSuchElementException;
import java.net.MalformedURLException;

import com.ms.com.Variant;
import com.ms.com.ComFailException;
import com.ms.com.SafeArray;

import ehubinterfaces.*;

/* Don't remove the comment about COM registration!! */

/**
 * This class is designed to be packaged with a COM DLL output format.
 * The class has no standard entry points, other than the constructor.
 * Public methods will be exposed as methods on the default COM
interface.
 * @com.register ( clsid=E3A53840-54D4-11D3-A8CA-00105A085D0A,
typelib=E3A53841-54D4-11D3-A8CA-00105A085D0A )
 */
public class SerialFileProducer implements
streamproducer.IHttpStreamProducer {

      private static final int BUFFER_START_SIZE = 4096;

      private static final int NO_MORE_DATA = 1;
      private static final int UNOPENED_PRODUCER = 2;
      private static final int PERMISSION_READ = 1;

      /* open()
       * This method prepares the handler for use.  It should be called
       * before fillBuffer().
       *
       */

    public SerialFileProducer()
    {
        m_Log = (IEhubLog)OLE.CreateObject("EhubObjects.Log");
        m_Log.setFacility("FileProducer");
        fileName = null;
```

```
        fileId = null;
    }

      public void open(String urlString) {

            if (openedSuccessfully)
                  return;

            compressionStreamIsOpen = false;
            initialURLString = urlString;
            headers = new HeaderVector(urlString);

            readBuffer = new byte[BUFFER_START_SIZE];

            try { builder = new CustomURLBuilder(urlString); }
            catch(MalformedURLException e) {
                m_Log.Log(EhubLogSeverity.ERROR, "Open - bad URL " +
urlString, 0);
                  // Later may throw an error
                  throw new ComFailException();
            }

        fileId = builder.lookupParameter(URLParameterEnum.ID);
        if (fileId == null)
        {
                m_Log.Log(EhubLogSeverity.ERROR, "Open - missing id in
URL ", 0);
            throw new ComFailException("URL does not contain the entry
id");
        }

            fileName = getFullFilePath(fileId);

        if (fileName == null)
        {
                m_Log.Log(EhubLogSeverity.ERROR, "Open - file with id "
+ fileId +
                        " not found", 0);
            throw new ComFailException("File not found in the
catalog.");
        }

        m_Log.Log(EhubLogSeverity.DEBUG, "Opening " + fileName + " [" +
fileId + "] " , 0);

            addContentLengthHeader();

            try { compressionLevel =
Integer.parseInt(builder.lookupParameter(URLParameterEnum.COMPRESSION_L
EVEL)); }
            catch(NumberFormatException e) {
                  compressionLevel =
CompressionEnumeration.NO_COMPRESSION;
            }

            try { fileStartPos =
Long.parseLong(builder.lookupParameter(URLParameterEnum.START_AT)); }
            catch(NumberFormatException e) {
                  fileStartPos = 0;
            }
```

```
        if (fileStartPos > 0)
                m_Log.Log(EhubLogSeverity.DEBUG, "Recovering from byte "
+
                        Long.toString(fileStartPos), 0);

            try {fileToRead = new RwwFile(fileName, fileStartPos); }
            catch(FileNotFoundException e) {
                m_Log.Log(EhubLogSeverity.ERROR, "Open - file not found
" + fileName, 0);
                    throw new ComFailException("File not found");
            }

            openedSuccessfully = true;

      }


      /* fillBuffer()
       * This method creates a buffer of data to send back to the
caller in the form of
       * a safe array embedded in a Variant.  The "count" argument is
the number of bytes
       * that the page suggests as a maximum, though it's remotely
possible that this could
       * be exceeded.
       *
       */

      public void fillBuffer(int count, Variant buffer) {

            if (!openedSuccessfully)
                  throw new ComFailException();

            if (count > readBuffer.length)
                  readBuffer = new byte[count];

            int numBytes = fileToRead.read(readBuffer, count);
            int before = numBytes;

      //    Debug.TRACE("read " + numBytes + " bytes");

            if (numBytes < 0) {

                  if (compressionStreamIsOpen) {
                        returnBuffer = closeCompressionStream();
                        numBytes = returnBuffer.length;
                  }
                  else
                        throw new ComFailException("End-of-stream");

            }

            else if (compressionLevel > 0) {
                  try { returnBuffer = doCompression(readBuffer,
numBytes); }
                  catch(IOException e) {
                        Debug.ASSERT(false);
                        throw new ComFailException("Compression
failed");
                  }
                  numBytes = returnBuffer.length;
```

```
            }
          else // no compression, so just copy the read buffer to the
return
          {
              returnBuffer = new byte[numBytes];
              System.arraycopy(readBuffer, 0, returnBuffer, 0, numBytes);
          }

              int after = returnBuffer.length;
       //     Debug.TRACE("Before: " + before + " and after: " + after);

       //     SafeArray safe = new SafeArray(Variant.VariantByte,
numBytes);
       //     safe.setBytes(0, numBytes, readBuffer, 0);
     //   buffer.putSafeArray(safe);
          buffer.putByteArray(returnBuffer);
       }


       /*  getHeaderAt()
        *  This method allows for the caller to determine what headers
the producer
        *  would like the http response to contain.  This method is
typically called
        *  in a loop.  The caller would call getHeaderAt with an index
 of 0, and then
        *  make a call with one, a call with two, and so on until
        *
        */

       public void getHeaderAt(int index, String[] name, String[] value)
{

              StringBuffer mutableName = new StringBuffer();
              StringBuffer mutableValue = new StringBuffer();

              try { headers.getHeaderAt(index, mutableName,
 mutableValue); }
              catch(NoSuchElementException e) {
                    throw new ComFailException("No header at index " +
index);
              }

              name[0] = mutableName.toString();
              value[0] = mutableValue.toString();

       }


       /* reportError()
        * A generic error-reporting mechanism.  Its behavior isn't set
 in
        * stone, but for now we just trace if debug mode is on.
        *
        */

       public void reportError(String msg)  {
            m_Log.Log(EhubLogSeverity.ERROR, "reportError - " + msg,
 0);
       }
```

```
      /*  close()
       *  This method closes the object and frees many of the resources
that
       *  it was using.  Of course, it can't be used in lieu of the
regular
       *  COM interface reference counting scheme.
       *
       */

      public void close() {

            Debug.ASSERT(!compressionStreamIsOpen);

        if ((fileId != null) && (fileName != null))
            m_Log.Log(EhubLogSeverity.DEBUG, "Closing " + fileName + "
[" + fileId + "] " , 0);

            if (!openedSuccessfully)
                  return;

            openedSuccessfully = false;

            try {
                  fileToRead.close();
            }
            catch(IOException e) {
                m_Log.Log(EhubLogSeverity.ERROR, "Close - error " +
e.getMessage(), 0);
            }
            fileToRead = null;

      }


      /* doCompression()
       * This method performs compression on the byte array that it
receives, returning
       * the compressed version.
       *
       */

      private byte[] doCompression(byte[] origArray, int numBytes)
      throws IOException {

            Debug.ASSERT(compressionLevel > 0);

            if (compressionLevel <= 0) {
                  Debug.TRACE("ERROR: Call to doCompression() with bad
compression level " + compressionLevel);
                  return origArray;
            }

            if (!compressionStreamIsOpen) {
                  compressionStreamIsOpen = true;
                  mos = new ManualOutputStream();

                  try { compOs = new GZIPOutputStream(mos, 256); }

                  catch(IOException e) {
                        Debug.TRACE("doCompression(): " +
```

```
 e.toString());
                  }


            }

            // byte[] choppedArray = new byte[numBytes];
            // System.arraycopy(origArray, 0, choppedArray, 0,
numBytes);

            try {
                  compOs.write(origArray, 0, numBytes);
            // compOs.write(choppedArray);
                  compOs.flush();
            }
            catch(IOException e) {
                  Debug.ASSERT(false, "doCompression() failed to write:
caught " + e.toString());
                  Debug.TRACE("ERROR in doCompression() failed to
 write: caught " + e.toString());
            }

            return mos.retrieveData();

      }


      /* closeCompressionStream()
       * This closes the compression stream and fills in the necessary
"closing" bytes.
       *
       */

      private byte[] closeCompressionStream() {

            if (!compressionStreamIsOpen) {
                  Debug.ASSERT(false);
                  return null;
            }

            try { compOs.close(); }
            catch(IOException e) {
                  Debug.ASSERT(false);
                  compressionStreamIsOpen = false;
                  return null;
            }

            compressionStreamIsOpen = false;
            return mos.retrieveData();

      }



      /* getFileName()
       * This method will return the full pathname of the file that
should be used by the
       * producer.
       *
     * J. Alvarado 7/19/00: removed code to use SERVER_PATH parameter.
This was
```

```
     * a security issue because it allowed the client applet to request
any file on
     * the server
       */

      private String getFullFilePath(String id) {

        IEhubCatalog    catalog;
        IEhubItem       fileEntry;
        File            actualFile;
        String          retVal = null;

        catalog =
(IEhubCatalog)(OLE.CreateObject("EhubObjects.Catalog"));
        fileEntry = catalog.Bind(id);
        // TODO: check for the type once we have type inheritance built
        // into the type path attribute

        // test for read permission
        if (((IEhubItemSec)fileEntry).Test(PERMISSION_READ))
        {
            actualFile = new
File(fileEntry.getAttribute("physicalLocation").getString());
                retVal = actualFile.getAbsolutePath();
        }

        return retVal;
      }


      /* addContentLengthHeader()
       * Adds the content length in as a header in the http response.
       *
       */

      private void addContentLengthHeader() {

            Debug.ASSERT(fileName != null);
            File semFile = new File(fileName + "_smf");

            if (!semFile.exists()) {

                File actualFile = new File(fileName);
                headers.addHeader(HeaderEnumeration.CONTENT_LENGTH,
String.valueOf(actualFile.length()));

            }

      }


      /* Private member variables */

      private String initialURLString;
      private CustomURLBuilder builder;
      private byte[] readBuffer;
      private byte[] returnBuffer;

      private boolean openedSuccessfully;

      private int compressionLevel;
```

```
    private ManualOutputStream mos;
    private DeflaterOutputStream compOs;
    private boolean compressionStreamIsOpen;

    private String fileName;
  private String fileId;

    private RwwFile fileToRead;
    private long fileStartPos;
  private IEhubLog m_Log;

    HeaderVector headers;

}
```

```
/*  TransactionController.java
 *  Author: John Abney
 *  Created: 6/29/99
 *  Last Modified: 7/23/99
 *
 *  This class represents a way to centrally control all ongoing
transactions.  The expected use
 *  of this class is for the client applet to create one instance of
this class and start all
 *  new transactions through the startTransaction() method.  The
signalStart(), signalStop(), and
 *  signalDestroy() methods should be called when start(), stop(), and
destroy() methods of the
 *  applet are called.  This allows the active TransactionHandler's to
be alerted.
 *
 */

import java.awt.*;

public class TransactionController {


      /* These intervals effect how long and how many times the
TransactionController will
       * wait for living threads to clean up their messes after the
Applet's destroy()
       * method is invoked.
       */

      private static final int DESTROY_WAIT_LENGTH = 1000;
      private static final int MAX_DESTROY_WAITS = 5;

      /* Public methods begin here */


      public TransactionController() {

            activeTransHandlers = new
ThreadGroup("activeTransHandlers");
            threadCounter = 0;

      }


      /*  startTransaction()
       *  This method causes a new transaction to be started on the
given URL.  It spawns a new
       *  thread to handle this transaction and then returns as soon as
possible.
       *
       */

      public synchronized void startTransaction(String urlString) {

            TransactionHandler transHandler = new
TransactionHandler(urlString);
```

```
            transHandler.start();

      }


      /*  signalStart()
       *  This method should be called from the applet's start()
 method.  This allows the
       *  signal to be passed down to the active TransactionHandler's.
       *
       */

      public synchronized void signalStart() {

            EventSignal sig = new
 EventSignal(EventSignal.APPLET_START);
            broadcastEventSignal(sig);

      }


      /*  signalStop()
       *  This method, like signalStart(), should be called from the
applet's stop() method.  This
       *  allows the signal to be passed down to the active
TransactionHandler's.
       *
       */

      public synchronized void signalStop() {

            EventSignal sig = new EventSignal(EventSignal.APPLET_STOP);
            broadcastEventSignal(sig);

      }


      /*  signalDestroy()
       *  This method should be called from the applet's destroy()
method.  This allows the signal to
       *  be passed down to the active TransactionHandler's.  Note that
this method checks all active
       *  threads to make sure that they aren't threads that
needCleanupTime().
       *
       */

      public synchronized void signalDestroy() {

            EventSignal sig = new EventSignal(EventSignal.APPLET_STOP);
            broadcastEventSignal(sig);
            int numWaits = 0;

            while ((numWaits < MAX_DESTROY_WAITS) && !allThreadsDone())
{

                  try { Thread.sleep(DESTROY_WAIT_LENGTH); }
                  catch(InterruptedException e) { }
                  numWaits++;

            }
```

```
	}


	/* Private methods begin here */


	/* allThreadsDone()
	 * This method cycles through the list of active threads and
checks to see if they're
	 * done executing.
	 *
	 */

	private boolean allThreadsDone() {

		int numThreads = activeTransHandlers.activeCount();
		Thread[] threadCollection = new Thread[numThreads];
		activeTransHandlers.enumerate(threadCollection);
		boolean needToWaitLonger = false;

		for (int i = 0; i < numThreads; i++) {

			if (
((TransactionHandler)threadCollection[i]).needsCleanupTime())
				return true;

		}

		return false;

	}


	/*  sendEventSignal()
	 *  This method signals a specified EventSignal to all active
TransactionHandler's.
	 *
	 */


	private void broadcastEventSignal(EventSignal sig) {

		int numThreads = activeTransHandlers.activeCount();
		TransactionHandler[] thArray = new
TransactionHandler[numThreads];

		try {
			for (int i = 0; i < numThreads; i++)
				thArray[i].signal(sig);
		}
		catch(ArrayIndexOutOfBoundsException e) {
			Debug.ASSERT(false, e.toString());
		}

	}


	public long threadCounter;	// NOTE: This counter is not meant
to be accurate.  It's intended to
```

```
                                                    // provide unique
names for debugging purposes.
      private ThreadGroup activeTransHandlers;

}
```

Appendix G

```
/*  TransactionHandler.java
 *  Author: John Abney
 *  Created: 6/29/99
 *  Last Modified: 8/20/99
 *
 *  This class represents the threads that are spawned by the
TransactionController.  Each
 *  is constructed with a given urlString.  It first creates the
 appropriate HttpStreamHandler
 *  and then initiates the transaction.
 *
 */

import netscape.security.PrivilegeManager;
import com.ms.security.PolicyEngine;
import com.ms.security.PermissionID;

import java.awt.*;

import java.net.URLConnection;
import java.net.MalformedURLException;
import java.net.URL;

import java.io.IOException;
import java.io.InputStream;

import java.util.Dictionary;
import java.util.Hashtable;


public class TransactionHandler extends Thread implements EventReceiver
{

      private static final int ONE_KILOBYTE = 1024;
      private static final int BUFF_SIZE = 64 * ONE_KILOBYTE;

      private static final int SLEEP_INTERVAL = 300;


      /* Public methods begin here */


      public TransactionHandler(String urlString) {

            Debug.ASSERT(urlString != null);
            originalURLString = urlString;

            handlerReady = false;
            handlerFactory = null;
            handler = null;
            finalURLString = null;
            urlStream = null;
```

```
            inNetworkRead = false;

            try { builder = new CustomURLBuilder(urlString); }
            catch(MalformedURLException e) {
                  // Debug.TRACE("WARNING: Caught MalformedURLException
in TransactionHandler()");
            }

            timeoutInterval = -1;       // Suitable default of no
timeout

      }


      /*  signal()
       *  This method receives signals from the applet and passes them
directly to the
       *  HttpStreamHandler's signal() method.  It does some extra
checks to insure that
       *  the messages can be appropriately received first.
       *
       */


      public synchronized void signal(EventSignal sig) {

            if (handler == null)
                  return;

            try {
                  Debug.TRACE("TransactionHandler.signal(): " +
sig.toString());
            // a client-not-interested signal would come from the
stream
            // handler, and should be used to abort the thread in case
            // it is blocked reading from the network.
            if (sig.getEventType() !=
EventSignal.CLIENT_NOT_INTERESTED)
            {
                        handler.signal(sig);
            }
            else
            {
                synchronized (this)
                {
                    if (inNetworkRead)
                    {
                        if ((handler != null) && !handlerIsClosed)
                        {
                            Debug.TRACE("Closing output file out-of-
band");

                            handler.close();
                            handlerIsClosed = true;
                        }
                    }
                }
            }
```

```
            }
            catch(Exception e) {      // Ignore for now, and ASSERT()
                  Debug.ASSERT(false, "signal() threw " +
e.toString());
                  Debug.TRACE("TransactionHandler.signal(): caught " +
e.toString());
            }

      }


      /*  run()
       *  This method overrides Thread.run() and represents the main
thread of
       *  execution.
       *
       */


      public void run() {

            if (Sysinfo.usesIESecurity())
                  assertIEPermissionsAndContinue();
            else if (Sysinfo.usesNetscapeSecurity())
                  assertNetscapePermissionsAndContinue();
            else {
                  Debug.ASSERT(false);
            }

      }



      /* needsCleanupTime()
       * This method is called to determine if this Thread is going to
need any cleanup
       * time.  It is not made thread-safe because this is an
unnecessary restriction, as
       * the NullPointerException is caught.
       *
       */


      public boolean needsCleanupTime() {

            long flags;
            boolean result = false;

            try {
                  flags = handler.getFlags();
                  if ( (flags &
HttpStreamHandlerFlagEnum.NEEDS_CLEANUP_TIME) != 0)
                        result = true;
            }
            catch(NullPointerException e) {
                  Debug.TRACE("TransactionHandler.needsCleanupTime(),
no handler exists");
```

```
                result = false;
            }

            return result;

        }

        /* prepareConnection()
         * This method gets everything set up to make the transfer of
data from the server
         * to the client.  It allocates a buffer, prepares the
SerialFileHandler, and then
         * connects to the now-updated URL.
         *
         */
        private void prepareConnection()
        throws IOException, MalformedURLException,
ClientNotInterestedException {

            Debug.TRACE("prepareConnection()");

            handlerFactory = new HttpStreamHandlerFactory();
            handler = handlerFactory.makeHandler(originalURLString);

         if (handler != null)
            handler.setEventReceiver(this);

            buffer = new byte[BUFF_SIZE];

            Debug.ASSERT(originalURLString != null);

            Debug.TRACE("Attempting to activate handler");
            finalURLString = handler.open(originalURLString);
         handlerIsClosed = false;

            if (finalURLString == null) {
                Debug.ASSERT(false, "Got null URL in
prepareConnection()!");
                return;
            }

            Debug.TRACE("***Attempting to create URL with " +
finalURLString);

            // Open the remote connection

            URL theURL = new URL(finalURLString);
            Debug.TRACE("***URL created");

            theConnection = theURL.openConnection();
         theConnection.setUseCaches(false);
         theConnection.setDoInput(true);
         theConnection.setRequestProperty("Content-Type",
"application/x-www-form-urlencoded");
         theConnection.connect();
            Debug.TRACE("***connection established");
```

```
        if (theConnection.getHeaderField(HeaderEnumeration.EHUB_LOGIN_FORM) !=
null)
        {
            // if we get back the login form interrupt the download and
tell
            // the user to re-login
                KillableFrame kf = new KillableFrame("Fakeout frame");
                  ModalOKDialog bokd = new ModalOKDialog(kf, "Session
Expired",
                                                         "Your login session has
expired. Please refresh your " +
                                                         "browser window,
login, and try the download again.");
                  bokd.show();
                  Debug.TRACE("prepareConnection() session has
expired");
            handler.signal(new
EventSignal(EventSignal.CONNECTION_NOT_ESTABLISHED));
            throw new ClientNotInterestedException();
        }
            handler.reportHeaders(getHeaders(theConnection));
            Debug.TRACE("***got headers!");

            urlStream = theConnection.getInputStream();
            Debug.TRACE("***Stream opened successfully");

      }



      private void assertNetscapePermissionsAndContinue() {

            PrivilegeManager.enablePrivilege("Netcaster");
            PrivilegeManager.enablePrivilege("UniversalExecAccess");

            connectionLoop();

      }


      private void assertIEPermissionsAndContinue() {

            PolicyEngine.assertPermission(PermissionID.SYSTEM);
            connectionLoop();

      }


      private void connectionLoop() {


            Debug.TRACE("New transaction started");

            KillableFrame kf = new KillableFrame("Fakeout frame");
            ModalOKDialog bokd;

            Debug.ASSERT(originalURLString != null);
```

```
            Debug.TRACE("run(): originalURLString is " +
originalURLString);
            if (originalURLString == null)
                  return;

            // First, open the connection to the remote server.

            try {
                  prepareConnection();
                  Debug.TRACE("Exitted from prepareConnection()");
            }
            catch(MalformedURLException e) {
                  bokd = new ModalOKDialog(kf, "Incorrect URL", "This
page contains an improper URL: " + e.toString());
                  bokd.show();
                  Debug.TRACE("Caught MalformedURLException in
run()!");
                  closeConnection();
                  if (handler != null)
                        handler.signal(new
EventSignal(EventSignal.CONNECTION_NOT_ESTABLISHED));
                  return;
            }
            catch(IOException ioe) {
                  if (handler != null) {
                        bokd = new ModalOKDialog(kf, "File Download
Error",
                                                "Could not download file.
Probably the file could not be found on the server or it is not accessible.");
                        bokd.show();
                        handler.signal(new
EventSignal(EventSignal.CONNECTION_NOT_ESTABLISHED));
                        Debug.TRACE("Connection problem, calling
closeConnection()");
                        ioe.printStackTrace();
                        closeConnection();
                  }
                  else {
                        Debug.TRACE("prepareConnection() failed to
create handler!");
                        Debug.ASSERT(false);
                  }
                  return;
            }
            catch(ClientNotInterestedException e) {
                  Debug.TRACE("Client signalled disinterest after
prepareConnection()");
                  closeConnection();
                  return;    // Anything else to do?
            }
            catch(RuntimeException e) {
                  Debug.TRACE("FATAL: " + e.toString());
                  e.fillInStackTrace();
                  throw e;
            }

            // Second, begin receiving data from the server.
```

```
            boolean clientWantsData = true;
            Debug.TRACE("Entering clientWantsData loop");
            while (clientWantsData) {

                  try {
                        transportData();
                        clientWantsData = false;
                  }
                  catch(ClientNotInterestedException cnie) {
                        Debug.TRACE("Client signalled CNIE, closing
connection");
                        closeConnection();
                        return;
                  }
                  catch(IOException e) {
                        handler.signal(new
EventSignal(EventSignal.CONNECTION_BAD));
                        Debug.TRACE("IOException, closing connection to
continue");
                        closeConnection();
                        if (handlerCapableOfRecovery())
                              clientWantsData = true;
                        else
                              clientWantsData = false;
                  }
                  /*
                  catch(RuntimeException e) {
                        Debug.TRACE("FATAL " + e.toString());
                        e.fillInStackTrace();
                        throw e;
                  }
                  */


                  if (clientWantsData) {
                        try {
                              Debug.TRACE("Attempting recovery at end
of loop...");
                              finalURLString =
handler.open(finalURLString);
                              urlStream = new
URL(finalURLString).openStream();
                        }
                        catch(Exception e) {
                              Debug.TRACE("Exception in recovery
attempt!!!" + e.toString());
                              return;
                        }
                  }

            }

            // Any other cleanup behavior?
            Debug.TRACE("Popped out of loop, closing connection");
            //try {
            closeConnection();
```

```
            //}
            //catch(RuntimeException e) {
            //    Debug.TRACE("FATAL: " + e.toString());
            //    throw e;
            //}

      }

      /* transportData()
       * This method returns when all of the data has been received
from the
       * remote URL, and performs the entire transportation process one
buffer
       * at a time.
       *
       */
      private void transportData()
      throws ClientNotInterestedException, IOException {

//          Debug.TRACE("transportData()");

            try {
                  while (blockForBuffer()) {
//                      Debug.TRACE("transportData(): Returned with
true from processBuffer()");
                  }
//                Debug.TRACE("transportData(): Returned with false from
processBuffer()");
            }
            catch(IOException e) {
            //    Debug.ASSERT(false,
"TransactionHandler.transportData(): " + e.toString());
                  Debug.TRACE("ERROR:
TransactionHandler.transportData(): " + e.toString());
                  throw e;
            }

      }


      /* blockForBuffer()
       * Reads a block of data from the remote url, if one is
available.  It then
       * writes out whatever data it received to disk.  It returns true
if it receives
       * data from the remote URL, and returns false if it receives the
EOF.
       *
       */

     private boolean blockForBuffer() throws
ClientNotInterestedException, IOException {

            int returnCode;

            if (urlStream == null) {
                  Debug.ASSERT(false, "urlStream was null in
```

```
processBuffer()!");
                    throw new IOException("urlStream was null in
processBuffer()!");
            }

            try {
            // if there are no bytes available, just block for the next
one,
            // otherwise get whatever is available and return.
            // NOTE: the code used to have the read(buffer) call
instead,
            // but on IE going through the proxy it was observed that
the
            // call would block until the whole buffer was available.
Also,
            // note that the available() call will block until some
data
            // is available, so this branch will not execute until the
            // stream is exhausted (will return 0 in this case).
            synchronized(this) {inNetworkRead = true;}
                  if (urlStream.available() == 0) {
                returnCode = urlStream.read(buffer, 0, 1);
                  }
                  else {
                returnCode = urlStream.read(buffer, 0,
urlStream.available());
                  }
            synchronized(this) {inNetworkRead = false;}
            if (handlerIsClosed)
                return false;
        }
            catch(IOException e) {
                  handler.signal(new
EventSignal(EventSignal.CONNECTION_BAD));
                  Debug.TRACE("blockForBuffer(): Signalling bad
connection");
                  // Sleep here?
                  return false;
            }

            if (returnCode != -1) {
//                Debug.TRACE("Read " + returnCode + " bytes");
                  try { handler.writeBlock(buffer, returnCode); }
                  catch(ClientNotInterestedException e) {
                        Debug.TRACE("blockForBuffer() passing
ClientNotInterestedException");
                        throw e;
                  }
                  return true;
            }
            else
        {
                  Debug.TRACE("blockForBuffer() got -1 from
urlStream.read()");
                  return false;
        }
      }
```

```
	private boolean handlerCapableOfRecovery() {

		if (handler == null) {
			Debug.TRACE("WARNING: handlerCapableOfRecovery()
called when handler is null");
			Debug.ASSERT(false, "handlerCapableOfRecovery()
called with null handler");
		}

		long flags = handler.getFlags();
		if ( (flags &
HttpStreamHandlerFlagEnum.DISCONNECT_RECONNECT_CAPABLE) == 0)
			return false;
		else
			return true;

	}


	/* getHeaders()
	 *  Produces a Dictionary containing all of the headers contained
in a given
	 *  URLConnection by cycling through them in a loop.
	 *
	 */

	private Dictionary getHeaders(URLConnection theConnection) {

		String currentKey, currentValue;
		Hashtable hash = new Hashtable();

		Debug.TRACE("***TransactionHandler.getHeaders()");

		for (int i = 1; ; i++)
	  {
			currentKey = theConnection.getHeaderFieldKey(i);
			if (currentKey == null)
				break;

			currentValue =
theConnection.getHeaderField(currentKey);
			Debug.ASSERT(currentValue != null);
			hash.put(currentKey, currentValue);
			Debug.TRACE("Header \"" + currentKey + "\" = \"" +
currentValue + "\"");
		}

		return hash;

	}


	/*  getTimeLimit()
	 *  Returns a long value corresponding to the number of
```

```
milliseconds that a
        *  connection must remain ideal in order to be considered timed
out.
        *
        */

      private long getTimeLimit() {

            if (timeoutInterval == -1)
                  return Long.MAX_VALUE;

            else
                  return timeoutInterval * 1000;

      }


      /* closeConnection()
       * Performs any cleanup on non-memory resources.  It closes the
HttpStreamHandler
        * and closes the urlStream.
        *
        */

      private void closeConnection() {

            Debug.TRACE("closeConnection()");

            try {
                  handlerFactory = null;
            if (!handlerIsClosed)
            {
                      if (handler != null)
                        handler.close();
            }
                  if (urlStream != null)
                        urlStream.close();
            theConnection = null;
            }
            catch(IOException e) {
                  Debug.TRACE("Yikes, closeConnection() failed!");
                  Debug.ASSERT(false);
                  // Anything else we can do???
            }

      }

      /* Private member variables */

      private HttpStreamHandlerFactory handlerFactory;
      private boolean handlerReady;
      private CustomURLBuilder builder;

      private int timeoutInterval;

      private HttpStreamHandler handler;
      private byte[] buffer;
```

```
  private String originalURLString, finalURLString;
private URLConnection theConnection = null;
  private InputStream urlStream;
private boolean inNetworkRead;
private boolean handlerIsClosed;

}
```